UCRL- 94376
PREPRINT

Developing a Portable Window/Menu Interface Using Object-Oriented Programming Tools

Margaret E. Poggio
Computer Systems Research Group
Electronics Engineering Department

This paper was prepared for submittal to
ACM Conference on Object Oriented Programming
Systems, Languages, and Applications
Portland, Oregon
September 29-October 2, 1986

April 1, 1986

Lawrence Livermore National Laboratory

Developing a Portable Window/Menu Interface Using Object-Oriented Programming Tools*

Margaret E. Poggio
Lawrence Livermore National Laboratory
P.O. Box 5504, L-156
Livermore, CA 94550
(415) 422-5453

Abstract

This paper will describe the motivation, approach, and results of an experiment using object-oriented programming tools to develop a portable window/menu system. This window/menu system was developed to provide a better interface for many existing computer programs as well as provide a tool set to use for new applications. Object-oriented programming tools were used to develop this system which was designed to run on ASCII-based terminals as well as bit mapped workstation displays.

*This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

Introduction

This paper will report on a project that used an object-oriented programming language to construct a portable window/menu system. The virtual window system encapsulates or hides the hardware window system dependencies from the user interface or menu system. Applications and tools will use menu objects to construct a uniform and consistent user interface. The menu objects utilize the virtual window objects. Initially a prototype of the entire user interface system was done using a specific hardware and software system but current work on the hardware dependent encapsulation issue is pursuing porting the tools to additional systems.

Being able to create independent windows (rectangular regions of a display) with the option of supporting occlusion, was seen as a powerful technique for displaying and manipulating data. The window virtual system initially targeted ASCII-based terminals but was to eventually migrate onto bit mapped workstation displays. The current system provides an interface that runs on ASCII terminals (currently any terminal that has VT100 terminal emulation). Work is now in progress to port the system to a VAXstation II exploiting its window system. As should be obvious from the fact that ASCII terminal types are supported, these tools are not graphical in nature. They do, however, support containment of textual material inside a bounded window area displayed on the physical terminal.

The menu system is built on top of the virtual window system. It provides a uniform way to present and control menus being used as an interface to a program or set of programs. These menus are easy to use and allow the user to brose various options of a program without having much prior knowledge of the program's functions. The advantages of having a uniform menu interface for a variety of tools are that a user may move from one tool to another with confidence in format of a menu, the meaning of information displayed in the menu, and the behavior exhibited by predefined actions of the menu system.

Motivation

The environment that motivated this work was that found in the Electronics Engineering Department of the Lawrence Livermore National Laboratory. Many large engineering codes used to model and analyze a wide variety of engineering problems already existed and new ones are continually being developed. Most of these codes are large pieces of FORTRAN that had little (if any) attention paid to their user interfaces. Even new programs being developed have little emphasis on human-computer interfaces. The programs are designed and written by people with little or no human-computer interface background. The major emphasis is usually on the accuracy of the algorithms that are being used to perform the desired function of the program. Also, many of these codes are used in a batch mode so the need for a user interface is deemed entirely unnecessary. Even codes that are run in a batch mode usually require input data that is frequently generated manually with a text editor. Often times the input data set is quite large and the format that is required makes the data set unintelligible to all but the most expert of users. Even something as simple as a form to fill in that builds a data set for a batch code would be an improvement for the new or inexperienced user.

Surprisingly complex codes are run on the some of the most sophisticated computers in the world with poor user interfaces that only a handful of people could learn to run the program. Many others whose work would be greatly aided by use of these tools have done without in order to avoid the painful learning process required to use the codes.

Although the extent of the problem is enormous, a small testbed project called EAGLES was started which would attempt to address several issues. The user interface aspect of EAGLES is the topic of discussion here.

Approach

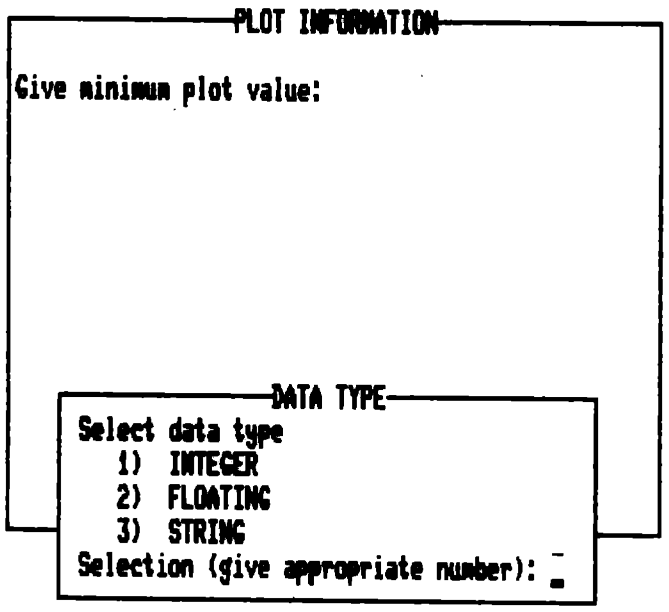
Object-oriented programming provides powerful support necessary to build user interfaces.[1,2] The idea of abstracting objects by defining classes that provided methods for manipulating the object fit well into the EAGLES model. To be able to move operator/operand interdependencies inside a class thus making a system more nearly independent of the objects they contained[3] was appealing. Also the class inheritance model provided by languages such as SMALLTALK-80[4] and Objective-C[5] would provide a convenient method for utilizing and creating user interface building blocks.

The initial development system was a DEC VAX/VMS system using VT100-type terminals. The eventual goal was to move the system to a DEC VAXstation II running VMS and/or to a UNIX-based workstation. All the workstations considered had bit mapped displays. Also, it was mandated that the system easily interface with the C programming language. Given these restrictions, Objective-C from Productivity Products International (PPI) was selected as the development language since it was available for both VMS and UNIX. Objective-C adds object-oriented constructs (modeled after SMALLTALK-80) to the C programming language and functions as a preprocessor for C.

Development of this window/menu system was the first practical experience that this author has had with object-oriented programming. The requirements were not well defined and extremely broad. The design process started with an attempt to define what the problems were and what would solve them. A major goal was to make the user interface as consistent and uniform as possible.[6] Input from various users and managers indicated a desire for a wide variety of interfaces that spanned the spectrum from sophisticated command languages (typically requested by experienced users) to iconical menus. It was decided that one interface would be selected and done well rather than attempting a couple of different types of interfaces (perhaps one for novices and one for experienced users) that would fragment the project and force both interfaces to not receive the proper attention in the design and implementation process. It is well known that a poorly designed user interface can lead to degraded user productivity, more user frustration, increased training costs, and usually a redesign and reimplementation.[7] The selection was made to provide an interface for the novice. The expert user could still use the old techniques, if desired, to use a code. An interface for novices would encourage many new users to try out codes. This was considered the most desirable alternative. EAGLES selected textual menus because some of the hardware was not capable of graphical display. The menus that were to be developed would be built on top of the more generic interface tool, the virtual window system.

Window System

The window system has been designed to establish a virtual window protocol for manipulation of textual windows. As mentioned earlier, this system is not attempting to deal with graphics. The window system for the VT100 compatible terminal is based on the Screen Management (SMG) routines provided for VAX/VMS version 4.0 and greater. These routines provide a variety of useful concepts and features for supporting window-like characteristics on an ASCII terminal. The concept of a pasteboard (one-to-one mapping with the physical device) is provided. The concept of a virtual display (window) is supported. An arbitrary number of virtual displays may be *pasted* (mapped) onto a pasteboard at specified locations. The SMG routines that support virtual displays allow for occlusion of displays, see Figure 1. Finally, the concept of a virtual keyboard is provided to allow virtual keyboards to be attached to specific virtual displays that are currently pasted to a pasteboard. The SMG routines allow for creation, manipulation, and deletion of these various items.



```
PLOT INFORMATION
Give minimum plot value:

DATA TYPE
Select data type
   1) INTEGER
   2) FLOATING
   3) STRING
Selection (give appropriate number): _
```

Figure 1. Two overlaid windows on an ASCII terminal.

At the beginning of the experiment, it was not known which bit mapped workstation was eventually to be used (perhaps more than one type), but the concepts provided by the SMG routines appeared to be a good foundation for developing the abstraction of this virtual window system with strong potential for conversion to a more sophisticated window system provided by a workstation.

Three low level classes were generated which supported the three major concepts (objects) of the SMG routines (and any good window system). One was for the pasteboard (the physical display), one for virtual displays (windows), and one for virtual input devices (e.g., keyboards). Since these classes directly called the SMG routines specific to VMS they were called VMSPasteboard, VMSDisplay, and VMSKeyboard. The Window class was developed to be the interface to all three of these low level classes. The Window class establishes the protocol that is used by the virtual window system. Each method in the Window class messages the appropriate lower level class(es) to accomplish the desired behavior. The Window class methods provide the interface to the low level classes and the methods in Window will be changed to interface with the low level classes for the bit mapped workstation window system. Another layer was added (the Panel class) as a subclass to Windows to provide a functional interface to Objective-C that would allow another level of abstraction above the Window class since the Window class was subject to radical changes within its methods. As the system works today, the need for this additional layer has proved unnecessary although it still remains. Figure 2 shows the class inheritance tree for the classes that support windows on a VT100 terminal using the VMS SMG routines. The class that appears in italics in the figure below is provided in the class libraries available from PPI with Objective-C.[8]

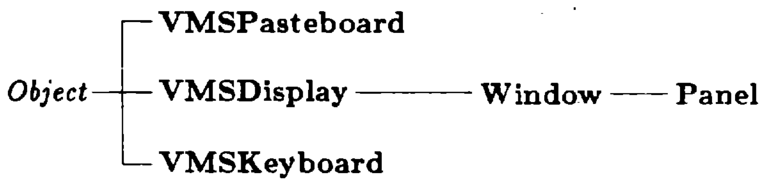


Figure 2. The inheritance tree for the classes that support VMS SMG VT100 windows.

The technique that will be used to support windows on various other devices (ASCII terminals and bit mapped displays) requires new low level classes that support the desired window system and a rewrite of methods in Window to message these new low level classes to achieve the desired functions. When the new classes are written they will replace VMSPasteboard, VMSDisplay, and VMSKeyboard (shown in Figure 2). The new classes (there will not necessarily be three) will provide the foundation for the appropriate window system.

The virtual window system has provided a concrete foundation for building tools that work well in a window environment. The menu system described in the next section is an example. Another example is a form system that has been developed to provide an additional user interface tool that allows user input in response to predefined prompts and other explanatory information that appears on a form. Each form appears in its own window. This mapping of one form to one window simplifies the way forms are displayed on the screen. Utilizing the classes designed to support windowing has proved to be useful for a variety of interface tools.

Menu System

Horizontal main menus that have vertical pull-down type menus (a la Apple's Macintosh) were decided upon as a model for this menu system (see Figure 3). Movement between menu items was to be accomplished by cursor keys in the case of the VT100-type terminal and a mouse will be used for the bit mapped workstation display.

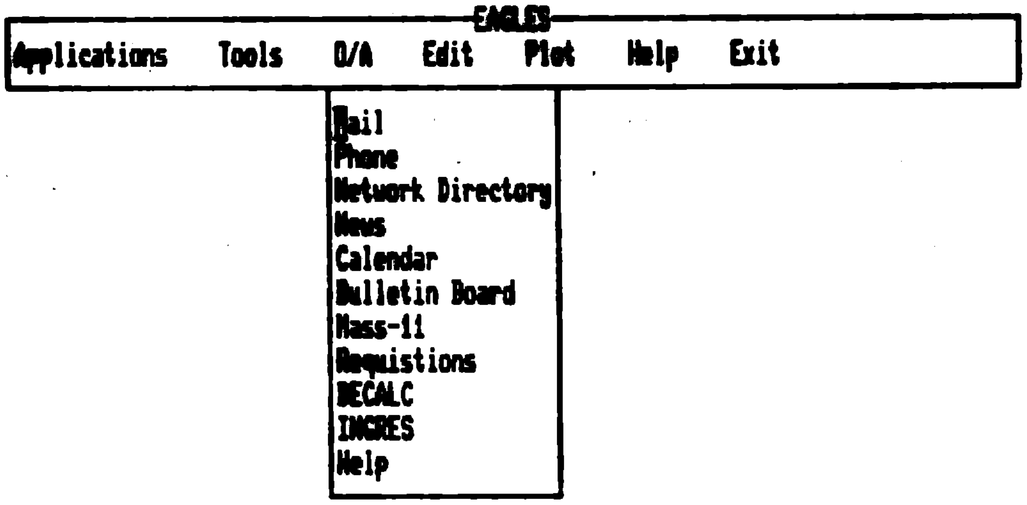


Figure 3. Horizontal and vertical menu.

The menu system creates the following classes: MenuItem, Menu, HorizontalMenu, and Vertical-Menu. MenuItem is the class which contains information about a single *button* (element) in a menu. This information includes what action should be taken if this button is selected. The Menu class is designed as a subclass of the class OrderedCollection (OrdCltn in Objective-C). This class provides most of the methods that are required to support all menus. It also provides the mapping for each menu onto its own window. This allows easy independent control of menus as they are displayed and removed from the screen. Two additional classes, HorizontalMenu and VerticalMenu, inherit Menu and respectively provide the additional methods that are unique to each of these menu types.

Figure 4 shows the inheritance tree of the classes developed to support menus. The classes shown in italics have been provided by PPI with Objective-C and the others have been newly created for this application.

Menus may be put together as a set that have a working relationship. For example, a horizontal

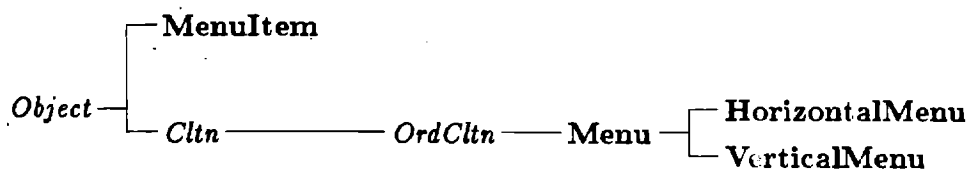


Figure 4. The inheritance tree for the classes that support menus.

menu typically has a vertical menu associated with each of its buttons. If one of these buttons is selected, the action associated with it is to display the corresponding vertical menu below it. The problem with this approach is that the stored information about a menu item knows nothing about the system identifier that gets assigned to a menu window at run-time. In order to get around this problem, an tree containing information on each menu is used at menu initialization to 1) create the set of menus for the application, 2) store the system window identifier associated with each menu, and 3) be traversed by each MenuItem whose action is to display another menu in order to get the window identifier for the menu it will display. Figure 5 shows the inheritance tree of the classes that support the tree and menu tracking facility just described. As before, italicized classes were provided with Objective-C.

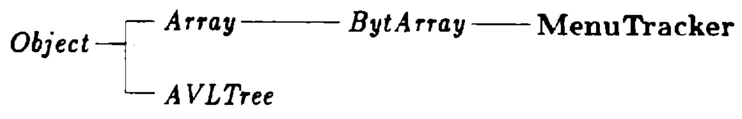


Figure 5. The inheritance tree for the classes that support menu initialization and tracking.

Several tools have been developed to utilize the various menu classes. First, a menu generator allows a program developer to easily generate a menu that will be used by an application. Another tool is the menu controller which controls menu manipulation, how menu selections are made, and the action that will be performed as a result of a button selection. The menu controller includes a help facility.

At menu creation, each button may be mapped to one of several action types. These actions include:

1. Display another menu,
2. Invoke the help facility,

3. Spawn a subprocess to perform another task or run another application,
4. Rє ırn a value to the application that called the menu controller, and
5. Exit back to the operating system.

All but item 4 from the above list are executed by the menu controller with no additional programming required to perform the indicated actions. Item 4 is provided so an application can have returned to it an integer value associated with a particular button. The application program can then take the appropriate action for the selected button.

By providing a standard set of actions and movement through a menu, the menu controller provides a consistent way for menus to function. This allows the menu interface to look familiar to the user across any tool or set of tools that utilize this menu system.

Results

To date, the proof of portability has not been established since a prerelease version of the underlying window management software that is needed for the VAXstation II has not yet been received. However, looking through the documentation for that software, it appears as though the windows and output to those windows should not be a very difficult task to interface to the established virtual window protocol. The more challenging task will be to map the input from the mouse on the workstation into the model for input (using cursor keys) that has been established in the current window design. The impact of the mouse input will probably trickle out of the input device class and into the input portion of the menu controller.

The virtual window system that has been developed to run on ASCII terminals works well. The variety of tools that have currently utilized this generic interface tool are indicative of its successful and flexible nature.

The ASCII terminal menu system has turned out well also. It has been successfully integrated into several different application programs at the time of this writing. The feedback from current users of programs that are using this menu system has been positive.

Conclusions

The classes and inheritance model provided by Objective-C has proved to be very powerful for developing the window and menu systems as well as various other interface tools (most of which have used the virtual window system classes as a foundation). This building block type of foundation that the class structure provides certainly indicates that object-oriented programming tools are well suited for this type of application. Regardless of any problems that have occurred during this experiment, the decision to use an object-oriented programming language has been unanimously considered correct.

Although an object-oriented programming language has proved to be useful for interface software, the use of such a language should not be allowed to replace any portion of the design of any system. Designing for an object-oriented language proves to actually be a more natural approach than many of the well-known common approaches such as Yourdon.[9] Unfortunately, the object-oriented design approach was not well understood at the beginning of this project. This experiment would have benefited from a more accurate and involved design process. Even though the design was not perfect, the software developed from this experiment is currently being used in a few applications with plans to expand it to several more.

One major problem that has probably resulted from the weakness of the design process has been how the hardware dependent classes and the virtual window class have been put together. The hardware dependent classes should probably be separated out of the inheritance tree for the virtual window class. This problem warrants more investigation into a better method for handling this situation. In any event, EAGLES has benefited from the user interface tools that have been described here.

References

1. Brad J. Cox, *The Message/Object Programming Model*, *Proceedings of SoftFair – A Conference on Software Development Tools, Techniques, and Alternatives*, IEEE Computer Society Press, July, 1983, pp. 51-60.

2. Brad J. Cox, *Message/Object Programming:An Evolutionary Change in Programming Technology*, *IEEE Software*, IEEE Computer Society Press, January, 1984, pp. 50-61.

3. Brad J. Cox, *Object-Oriented Programming – A Power Tool For Software Craftsmen*, *UNIX Review*, February/March, 1984.

4. Adele Goldberg and David Robson, *SMALLTALK-80 – The Language and Its Implementation*, Addison-Wesley, 1983.

5. Brad J. Cox, *Object-Oriented Programming in C*, *UNIX Review*, October/November 1983, pp. 67-70.

6. Reid G. Smith, Gilles M. E. Lafue, Eric Schoen, and Stanley C. Vestal, *Declarative Task Description as a User-Interface Structuring Mechanism*, *IEEE Computer*, IEEE Computer Society Press, September, 1984, pp. 29-38.

7. James D. Foley, Victor L. Wallace, and Peggy Chan, *The Human Factors of Computer Graphics Interaction Techniques*, *IEEE Computer Graphics and Applications*, IEEE Computer Society Press, November, 1984, pp. 13-48.

8. —, *Objective-C Reference Manual*, Productivity Products International, 1984.

9. Grady Booch, *Object-Oriented Development*, *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, IEEE Computer Society Press, February, 1986, pp. 211-221.